

Providing Dynamic Network Information to Distributed Applications

by

Gan Chen

May 2001

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science
under the instruction of Dr. Masha Sosonkina

Department of Computer Science
University of Minnesota
Duluth, Minnesota 55812
U.S.A.

Abstract

The majority of distributed applications are unable to learn and predict network conditions at the run-time and thus their performance suffers from the inability to react to changes in network performance. By providing applications with network information and letting applications adapt themselves to the network conditions is a method for solving this problem.

In this thesis, we design and implement a simple tool – Network Information Collection and Application Notification (NICAN). During the life-time of an application NICAN collects various network performance characteristics based on application requests and provides this information to the application. NICAN is designed in a modular way: network information collection, application notification and data analysis, history data table, and an interface to the application. The functionality of NICAN can be easily enhanced by adding more modules. Experimental results show that our tool can be used by an application developer for performance tuning. It can interface with distributed scientific applications and handles heterogeneous computing platforms.

Acknowledgments

Special thanks to Dr. Masha Sosonkina, who guided this research work and helped me whenever I was in need. Her easy grasp of network and distributed computing at their most fundamental level helped me in the struggle for my own understanding.

I would like to express my gratitude to the two other members of my examination committee, Dr. Ted Pedersen and Dr. Robert McFarland, for their patience and support.

I would also like to thank all the knowledgeable and friendly faculties and fellow graduate students for their support.

Dedication

To my wife, Shirley Xiaochun Liu, and my parents, Xuehua Tang and Bilian Chen, for their endingless support and encouragement.

Contents

1	Introduction	1
1.1	Network-aware Application Development	2
1.2	Proposed Thesis Work	4
1.3	Related Research	4
1.4	Thesis Organization	6
2	Background	7
2.1	SNMP - Simple Network Management Protocol	7
2.1.1	SNMP Architecture	8
2.1.2	SNMP Message Standard	9
2.1.3	Structure of Management Information	9
2.1.4	Management Information Base	10
2.1.5	NET-SNMP Project	11
2.2	Network Bandwidth and Latency	12
2.3	Network Protocol Layers	14
3	Design of NICAN	16
3.1	The Architecture of NICAN	16
3.2	Network Information Collection	18

3.3	Application Notification and Data Analysis	20
3.4	NICAN Interface to Application	20
3.5	Historical Data	22
3.6	The Big Picture	23
4	Implementation of NICAN	24
4.1	NICAN Interface to Application	24
4.1.1	The Application Request	25
4.1.2	Shared Memory	28
4.1.3	Interprocess Communication	29
4.2	Network Information Collection	30
4.2.1	Information Provided by SNMP	31
4.3	Application Notification	34
4.3.1	Signaling Mechanism	34
4.4	Historical Data Table	36
4.5	NICAN Architecture Revisited	36
5	Experimental Results	38
5.1	NetPIPE over TCP protocol	40
5.2	NetPIPE over MPI	44
5.3	Experiments for Heterogeneous Computing Platforms	47
6	Conclusions and Future Work	50
6.1	Future Work	51

List of Figures

2.1	SNMP Architecture.	8
2.2	OSI Management Information Tree.	10
2.3	The MIB Subtree.	11
2.4	The Internet Architecture.	14
3.1	NICAN Architecture.	17
3.2	NICAN Information Collector.	18
4.1	Pseudocode for NICAN interface.	26
4.2	Application Request Data Structure.	26
4.3	Shared Memory Structure between Application and NICAN Interface.	29
4.4	NICAN Processes Application Request.	30
4.5	Network Information Collection.	31
4.6	Processor Time for NICAN with and without polling pause.	33
4.7	NICAN History Table Structure.	35
4.8	Add Data to NICAN History Table.	35
4.9	NICAN Architecture in Details.	37
5.1	NetPIPE Ping-Pong Test Pseudocode.	39
5.2	Interconnection of The Hosts Used in The Experiments.	39

5.3	Simultaneous NetPIPE (TCP) and NICAN throughput measurements: between Hosts H1 and H3.	41
5.4	Simultaneous NetPIPE (TCP) and NICAN throughput measurements: between Hosts H2 and H3.	41
5.5	Performance of NetPIPE (TCP) with and without invoking adapta- tion mechanism: throughput measurements between Hosts H2 and H3.	43
5.6	Performance of NetPIPE (TCP) with and without invoking adapta- tion mechanism: time to transmit a message between Hosts H2 and H3.	43
5.7	Latency measured by NICAN between hosts H2 and H3.	44
5.8	Simultaneous NetPIPE (MPI) and NICAN throughput measurements: between Host H1 and H3.	46
5.9	Performance of NetPIPE (MPI) with and without invoking adapta- tion mechanism: throughput measurements between H1 and H3. . .	46
5.10	Performance of NetPIPE (MPI) and NICAN throughput measure- ments with competitor NetPIPE (TCP): between H1 and H3.	47
5.11	Simultaneous NetPIPE (TCP) and NICAN throughput measurements: between Hosts H1 and H4.	48
5.12	Performance of NetPIPE (TCP) with and without invoking adapta- tion mechanism: throughput measurements between H1 and H4. . .	48

Chapter 1

Introduction

Many modern applications benefit from being run on networked systems. However, it is a challenging problem to make efficient use of network resources. The network resources available for an application are not only affected by the application itself, but also by all the other applications that are using the network at the same time. Application performance suffers from the inability to learn the network conditions at runtime and thus the inability to react to changes in network performance. This is critical for scientific applications which usually perform computation-intensive tasks on a distributed system. An application can reserve resources it needs in a reservation-based network in order to improve its performance, for example, as in a grid system [4]. But not all modern networks support reservations and those reservation-based systems have the drawback of high overheads [6]. Most of the networks are of the “best effort” type. A “best effort” network tries to deliver packets as soon as possible but provides no guarantees of services [13]. In the absence of network reservation capabilities one way to solve this problem is to make applications network-aware. Based on the knowledge of the current network conditions,

network-aware applications try to adapt themselves by changing their demands of network resources so that they can perform communication more efficiently. For example, instead of suffering from the low bandwidth due to network congestion and risking losing data, an application may reduce its request of bandwidth. In general, application adaptation mechanisms depend on the type of application and the network information available. A scientific application may perform more local computations while waiting for the data from the remote hosts [20]. A file transfer application may choose a server with a better connection as an adaptation step [9]. Adaptations are possible if applications receive timely information about the network, either by probing the network or interfacing with a tool that delivers this information to applications.

1.1 Network-aware Application Development

A network-aware application has two essential components: the application itself which performs a specific task and adapts to the network conditions, and the network information provider which collects the network information and delivers it to the application. The former requires a high level of abstraction and problem domain knowledge while the latter requires a knowledge of the low-level network characteristics. This complicates the task of developing network-aware applications since the developer has to be familiar with both application domain and network programming and analysis domains.

Developers can let applications use implicit network feedbacks. An example of such a feedback is that of the perceived packet loss indicating possible congestion. Alternatively, developers can let applications use benchmarks to probe the

network. Both these solutions have significant drawbacks that restrict their usefulness: overhead from the benchmark computing and extra traffic on the network are introduced; the former requires the extensive knowledge of networking for an application developer. What is more detrimental is that applications need to be modified to incorporate these solutions, so the functionality of the applications is affected.

A more acceptable solution is to provide application developers with an application programming interface (called NAAPI) which supplies the network information upon request. Thus an application programmer can ask the NAAPI to provide the service that the application may need without knowing the underlying network details and without many modifications of the application itself.

Another reason an NAAPI is desirable is the existence of a variety of network characteristics that different classes of applications might request. One application may require very high peak bandwidth while it can tolerate a low average bandwidth. Another application may be sensitive to the average bandwidth while accepting a relatively high latency. Thus a network-aware application has to have the option of requesting particular network information, i.e., an ability to choose the type of information that is critical to the application. By utilizing the critical network information, the application can adapt to the network conditions in a more effective way.

A well-designed NAAPI should be able to collect a rich set of network information depending on the application requests. It should also possess an efficient

mechanism for interacting with an application. The interaction mechanisms used most commonly are query-based and call-back approaches (see section 3.3). NAAPI hides the details of the underlying network and thus makes application developing easier and more efficient.

1.2 Proposed Thesis Work

In this thesis, we design an easy-to-use tool, Network Information Collection and Application Notification (NICAN), that collects various information on network conditions and notifies applications of the occurrence of the specified events. Such network events as reaching certain bounds on bandwidth and latency can be monitored. Other events can be easily specified by the application. This event mechanism does not need to interrupt application execution. Upon occurrence of specified events, the application can choose to ignore the events or adapt itself to the current network condition. The network information collection introduces no overhead into the network communication. The modular design of NICAN enables the enhancement of the functionality of NICAN with more options for network information collection. The proposed NICAN interface and notification mechanism require minimal modifications of the applications.

1.3 Related Research

Much research has been done in the field of collecting network performance information and analyzing it. NetLogger (Tierney, Johnston, Crowley, Hoo, Brooks and Gunter [25]) is a proposed methodology that collects large amounts of network event information, logs the events, and uses an agent-based system to manage the data.

An application can benefit from this large amounts of data analysis. However, the logging and analysis typically produce megabytes of data in a short period. Thus, NetLogger incurs a large overhead on memory and input-output resources, which can be important to computationally intensive applications. Scion (Norton and Adams [12]) from the NetSCARF team collects and analyzes network data from routers. However, Scion lacks the interface to the network application. To utilize the information provided by Scion, the application developers must build their own interface for gathering the information that their applications need.

Remos [3, 8] is a query-based interface for network-aware applications. It allows the application to utilize the network environment information at runtime and to interact with the Remos interface to adapt to this environment. The drawback of Remos is that it requires multiple modifications of the application codes. The application programmer must let the application call the interface functions explicitly to perform the interaction and stay idle while waiting for the function return. Another consideration is that Remos assumes a fixed value for per-hop delay, the direct link between two network nodes. Although latency usually changes little during an application's life-time, changes do happen, especially when network congestion occurs. The inaccurate measurement of latency can affect the performance of Remos.

Network Weather Service (NWS) [26] predicts network capacity (throughput and latency) and CPU load based on monitored performance, and forecasts the information to the scheduler of a networked computational system, which in turn uses this information for application scheduling. NWS is used in a metacomputing environment such as a grid [4].

NetPIPE [18] is a benchmark application for measuring network protocol performance. It can be used in various types of networks. NetPIPE uses the ping-pong transfer mechanism which transfers only one data block at a time from each node to each other node using point-to-point connection, and calculates the bandwidth based on the message size and the time it takes to transfer the block. NetPIPE is used in this thesis as an application code to test NICAN.

1.4 Thesis Organization

Chapter Two provides the background material, which includes definitions, such as bandwidth and latency, an introduction to network management systems (SNMP), and an overview of network protocol stacks. Chapter Three presents the rationale for Network Information Collection and Application Notification (NICAN) design and its modular architecture. Chapter Four explains in detail the implementation of NICAN. A more abstract description of design and implementation of NICAN can be found in [21]. The experimental results are summarized in Chapter Five. Conclusions are made in Chapter Six, followed by remarks for future work.

Chapter 2

Background

In this chapter, the background material for this thesis is presented. The first section introduces the Simple Network Management Protocol (SNMP) which is used in this thesis for collecting network information. This section covers the basics of SNMP. The second section provides information about network characteristics, bandwidth, and latency. This section also covers the definition and measurement of bandwidth and latency. The information about the network congestion control is also presented. The last section introduces the Internet Architecture view of network protocol layers.

2.1 SNMP - Simple Network Management Protocol

The description of SNMP is based on Subramanian's Network Management: Principles and Practice [24].

The Simple Network Management Protocol (SNMP) was developed in 1988 by Internet Engineering Task Force (IETF) and has gained widespread acceptance since 1993. As its name indicates, SNMP is a simple solution for network management.

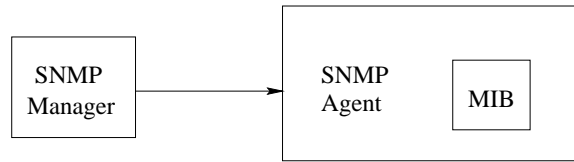


Figure 2.1: SNMP Architecture.

Simplicity and extensibility are the main advantages of SNMP. With its simplicity, SNMP is easy to implement and vendors can easily build SNMP agents for their products. SNMP agents can be easily extended to cover device-specific data. SNMP has become the de facto standard for internetwork management. It is now the most widely used network management system. Most of today's network nodes have built-in network agents which can respond to a poll from an SNMP network management system.

2.1.1 SNMP Architecture

SNMP defines the client/server relationship. It has two essential elements: a client program (called the network manager) and a server program (called the SNMP agent). The agents are the entities which have interfaces to the managed devices containing managed objects. A database, the management information base (MIB), is controlled by an SNMP agent and arranges a set of statistical and control values of the managed objects. Managers access the managed objects by communicating with the agents. Figure 2.1 shows the one manager - one agent model of SNMP. In practice, one manager can manage multiple agents, and one agent can be accessed by more than one managers.

2.1.2 SNMP Message Standard

SNMP has five primitive operations: get, get-next, get response, set, and trap. A “get” is issued by the manager to get the value of a named object. A “get-next” is used by the manager to get the name and value of the “next” object of the given SNMP object. A “set” is used by the manager to set a named object to a specific value. The agent sends a response message in response to the get, get-next or set requests. A “trap” is used by the agent to send an event notification to the manager in the case of a problem other than any polling of the device.

2.1.3 Structure of Management Information

The Structure of Management Information (SMI) [15] is used to organize the managed network objects. The SMI gives the object name and describes the information of the managed objects so that the objects can be logically accessible. Logical accessibility of an object means that it is stored somewhere and can be retrieved and modified in some way. An object name given by SMI is also called object identifier (OID) and is the unique identifier of the object. The data type of an object is defined by the Abstract Syntax Notation One (ASN.1). The Basic Encoding Rules (BER) describes how the managed information is serialized for transmission between machines.

Managed objects are defined in a tree structure called a management information tree (MIT). Figure 2.2 shows the OSI MIT. The number in each circle is the designation of the associated object. There is no explicit designation for the root node. Except for the root node, each node is under some node at higher level.

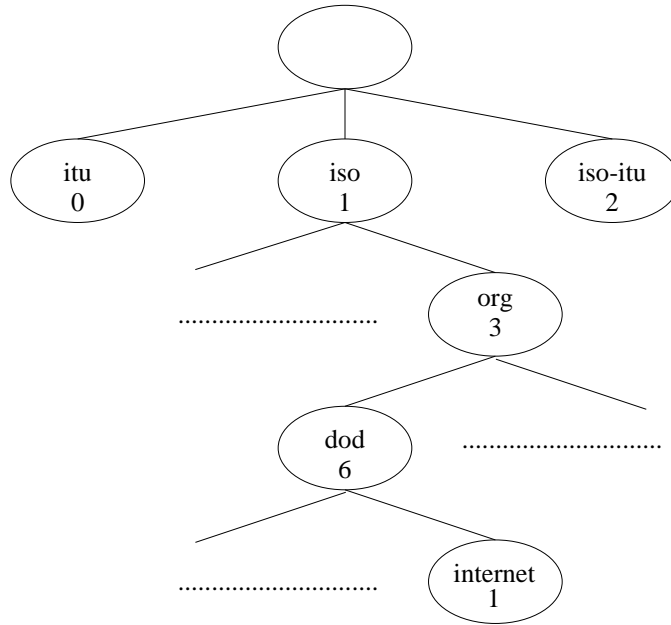


Figure 2.2: OSI Management Information Tree.

Each managed object has its own unique object identifier, or name, which is written as a sequence of integers separated by periods. The integers are actually the designators of the objects as describe above. For instance, the sequence .1.3.6.1.2.1.1.3.0 specifies the object *sysUpTime* within the system group which is in the *mib* subtree which in turn is in the management subtree.

2.1.4 Management Information Base

Management Information Bases (MIBs) define the properties of the managed objects of the managed device. Each such device keeps a database of values for each of the definitions written in the MIB. SNMP manageable objects are defined in MIB-II [14] and are grouped in the *mib* subtree in MIT. Totally, *mib* subtree contains ten groups: *system*, *interfaces*, *at*, *ip*, *icmp*, *tcp*, *udp*, *egp*, *transmission*, and *snmp*.

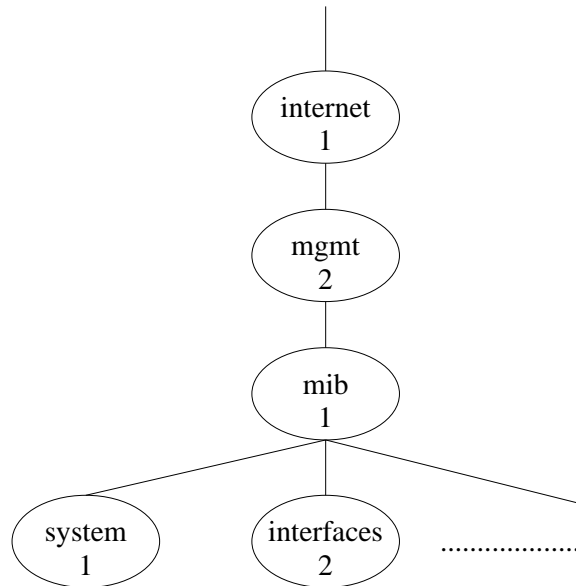


Figure 2.3: The MIB Subtree.

Figure 2.3 only shows the part of the *mib* subtree and contains only the *system* and *interfaces* groups. From it we can see that the *mib* group is the subtree of the *management* node, which in turn is the subtree of the *internet* node.

2.1.5 NET-SNMP Project

NET-SNMP project [10] is based on the SNMP implementations from Carnegie Mellon University (CMU) and University of California at Davis (UCD). The UCD-SNMP project was based on CMU SNMP implementation. On October 2000, the UCD-SNMP was renamed NET-SNMP.

The network management programmers can easily use the portable library provided by NET-SNMP to write their own code to manage any manageable network

objects through the SNMP agents, which are also provided by NET-SNMP.

2.2 Network Bandwidth and Latency

In a wide variety of networks, the two most important parameters are network bandwidth and latency. Network bandwidth measures the capacity of the network link, measured in bits per second (bps). It is common to distinguish “physical” bandwidth, which is determined by the physical properties of the network components, such as material properties for network connections, number of hops and switches, and “effective” bandwidth or throughput, which changes dynamically with the network load and the actual routing of a message. We can get the value of effective bandwidth if we know how much data goes through a link in a particular time period.

$$\text{bandwidth} = \frac{\text{number of bits transmitted during time period } t}{t} \quad (2.1)$$

Latency represents the delay of a link, i.e., how long it takes for a data unit to reach destination. Latency is usually measured in round-trip time (RTT) rather than one-way delay since in most cases we would like to receive the feedback from the destination after it is received. Latency includes the propagation delay, transmission time for the data unit, and the queue time if any [13]. A data bit cannot be transferred at a speed that exceeds the speed of the light. So, propagation delay is affected by the distance between the sender and receiver. The transmission time is the total time for transferring the data unit across a link. It is affected by the size of the data unit and the network transfer capacity, which is the bandwidth. If a data unit is queued before it is actually forwarded to the next hop, the queue time should also

be included. In general, the total latency is

$$Latency = Propagation\ delay + Transmission\ time + Queue\ time \quad (2.2)$$

In practice, latency can be measured by recording the RTT for sending a small message to remote host and waiting for the remote host to echo the message. The utility programs `ping` and `traceroute` can give more accurate and dynamic results.

Different applications have different requirements on network performance characteristics. For example, a video application transfers video information as frames of data, where each frame can have 128 KB data. In order to have high-quality video on the remote end, the application needs to transfer 30 frames per second, which requires a throughput of 31.6 Mbps. Throughput higher than this rate does not help a video application [13]. But if throughput is too low, the quality of the video may be unacceptable.

A severe problem of network performance is congestion. Network congestion occurs when too many packets are pushed into the network and overflow a router buffer. The problems with congestion are that the congested router has to drop message packets from all the communicating applications and that congestion spreads very rapidly since dropped packets are retransmitted. Thus network efficiency drops severely. Network throughput decreases while latency grows rapidly. Flow control and congestion control mechanisms are used to deal with buffer overflow problems. Flow control is used by the receiver to keep the sender from sending more data than the receiver can handle. Congestion control prevents too much data from being put into the network and overflowing the switches or routers. The difference between flow control and congestion control is that the former is an end-to-end problem while

Applications (e.g. MPI)	Applications (e.g. SNMP)
TCP	UDP
IP	
Data Link	

Figure 2.4: The Internet Architecture.

the latter concerns network nodes interactions.

2.3 Network Protocol Layers

A network is designed in an abstract and layered way such that each lower layer encapsulates some important aspects of the network system, hides the implementation details, and provides an interface to the upper layer [13]. Figure 2.4 shows the Internet Architecture view of the network protocol layers. This architecture is also referred as TCP/IP architecture. The lowest layer, Data Link layer, implements the combination of hardware and software and provides the interface between software and hardware. Ethernet [1] or Fiber Distributed Data Interface (FDDI) [7] are common protocols in Data Link layer. The Internet Protocol (IP) layer defines the Internet addressing scheme and transfers information between the Data Link layer and Transport layer. The third layer is the transport protocol layer

containing two different protocols, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP provides reliable, connection-oriented byte-stream data transmission, while UDP provides unreliable, connectionless message transmission. Application Protocols are written on top of TCP or UDP. For example, SNMP is an application protocol that uses UDP for communication, and Message Passing Interface (MPI) [5, 19] is an application protocol that uses TCP.

Message Passing Interface Standard (MPI) is a message-passing system which sits on top of the TCP layer. MPI is used by applications to perform parallel computing. MPI abstracts the underlying network configuration and provides applications with a standardized portable interface. Most of the distributed scientific applications use MPI for interprocess communication.

Chapter 3

Design of NICAN

In this chapter, the detailed design of Network Information Collection and Application Notification (NICAN) is described. This includes the NICAN structure, the essential components and the functionality of NICAN, and the design requirements.

3.1 The Architecture of NICAN

Network Information Collection and Notification (NICAN) has four essential components:

1. The information collector, which collects the network information requested by the applications.
2. The component for data analysis and application notification. This component analyzes the network information and sends out a signal along with the information which is helpful to the application.
3. The interface to the network applications. The interface component consists of a few function calls inserted into application to interact with NICAN and

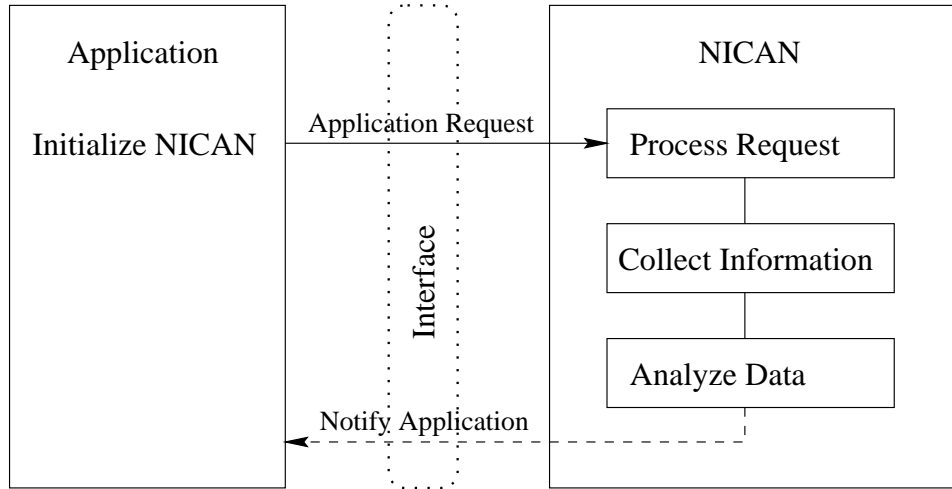


Figure 3.1: NICAN Architecture.

capture the signal from NICAN.

4. The historical data component. The historical network information that NICAN captures is recorded into a table to aid analysis and to predict future performance.

In NICAN, each component is separated from the others and is encapsulated into a module that can be chosen depending on the type of network, network software configuration, and the information to be collected. Figure 3.1 shows the general outline of the architecture of NICAN. An application uses an NICAN interface to send its request. The NICAN processes the request and then uses the network information collector to collect the requested information. Collected data is analyzed and notification is sent from NICAN to the application if some network event happens. The network collector and data analysis components use submodules to perform the particular tasks. The modular design makes NICAN extensible. NICAN function-

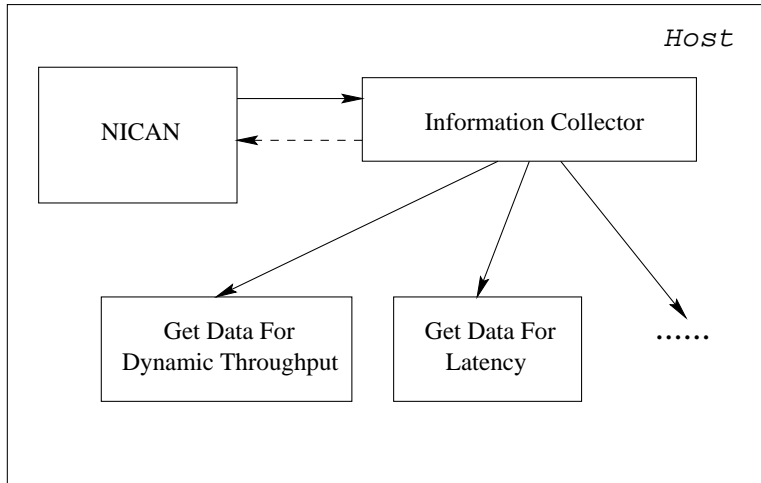


Figure 3.2: NICAN Information Collector.

ality can be very easily enhanced by attaching a new module to NICAN to perform the new information collection or new data analysis.

3.2 Network Information Collection

Network information is collected at the endpoints where the application is run. At this stage, the type of the information collected by NICAN is such that network dynamic throughput and latency can be easily calculated. These network characteristics are typically most vital for distributed applications. Other parameters, such as the number of TCP retransmission packets, can also be obtained by NICAN. Figure 3.2 shows the network information collection for NICAN. NICAN represents all other modules except the collector. The collector module consists of submodules for performing the specific information collection tasks. It has a module for getting dynamic throughput, another module for measuring latency and if needed, other

modules for other information. The solid arrow line from NICAN to the collector indicates that NICAN spawns the collector, and the dashed arrow line from the collector to NICAN shows the return information flow.

The first consideration of the information collector is that it should not significantly reduce the throughput, which means that the collector should not affect the execution of the application or the communications. The only exception is that in some networks, we may need to use benchmarks (ping-pong test [18]) to collect network information when there is no other option. However, benchmarking is very costly and cannot be accomplished dynamically without affecting the execution of the application. Thus the preferred method of throughput collection is to use SNMP protocol [24]. Benchmarking can be done statically.

We opted for separate collection of the information for different network characteristics because the information collection must be collected in a timely fashion and different collection submodules need different amounts of time. Whenever an information value becomes available, NICAN processes this information immediately without waiting until other informations becomes available. This is important since network condition can change quickly. Information process delay would affect the performance of NICAN and would lead NICAN to give out useless notification or even wrong directions for application adaptation. In particular, the throughput collection is independent of the latency collection as can be observed in Figure 3.2. This design enables an easy augmentation of the collection process with new options. This ensures its applicability to a variety of network interconnections and network software.

3.3 Application Notification and Data Analysis

The two design options for network-aware application programming interface are query-based [3, 8] and call-back mechanisms [11]. The query-based mechanism requires that an application explicitly requests the interface to provide the information, which causes the application to wait for the result from the interface. The call-back mechanism decouples the network analysis from application execution. With the call-back approach, applications need not know how the interface is implemented nor how it works. The application receives system signals sent by NICAN when some network environment condition changes.

NICAN notification is based on the call-back method. A signaling mechanism is needed for this method. NICAN delivers signals to the applications along with the information whenever this information (e.g., throughput or latency or both) falls outside a certain range. This range can be requested by the application at startup time or chosen from a default range by NICAN. In some cases, when the value of a parameter stays unchanged or the change is small enough to be ignored for a while, NICAN sends a signal to indicate that this network performance characteristic has become stable.

3.4 NICAN Interface to Application

The NICAN interface is the communication channel between an application and NICAN. The interface provides an application with an entry function which sends the application request to NICAN. The request includes the communication characteristics and possibly their boundary values. We note that if some other conditions

occur in the network, NICAN can notify the application as well, i.e., we should not assume that the users know the bounds. These parameters are passed from the application to NICAN via the interface. The interface also captures the signals sent by NICAN. Upon receiving these signals, the interface helps to deliver the information provided by NICAN to the application by using signal handlers.

Two approaches can be used to accomplish the event handling required by NICAN. One is to use multi-signals. In this approach, each network performance characteristic needs one unique signal corresponding to it. An application can redefine the default behaviors of the two available user signals. The limitations of this approach are obvious: the number of available signals can not satisfy the requirement of both the application and NICAN, even though some signal handlers can be changed and then be utilized by the application. Another approach is to use one signal to handle all information notifications of NICAN. Upon receiving the signal, the signal handler processes and analyzes the information sent from NICAN. Based on the type of information the application receives, it then adapts the network appropriately. This approach simplifies information processing and application adaptation. The choice of the single signal handler approach also satisfies the modular design of NICAN since more information handling can be added easily which does not affect other components of NICAN.

The role of the interface is to assist the interaction between an application and NICAN. The design of the interface satisfies one of the NICAN design requirements which is not to affect application execution. An application that uses NICAN to aid its performance can detach NICAN at any time during its execution by sending

another request to the NICAN interface.

An obvious requirement for such an interface is that it be lightweight and able to perform an interaction between an application and NICAN efficiently. Thus we decided to design the interface using the shared memory paradigm and interprocess communication (IPC). Shared memory provides an efficient way for processes running on the same host to exchange or share information. Shared memory is very suitable for two processes that are related to each other, such as two processes having a child-parent relation, which means the child process is produced by the parent process during its execution. Shared memory can be created dynamically and can be destroyed after it is used. IPC is another way for two or more processes running on one host to communicate with each other. Some IPC mechanisms, such as first-in first-out pipe (FIFO) and Unix-domain network socket interface, allow any two processes running on one host to communicate with each other.

3.5 Historical Data

A historical record of the collected information needs to be saved for analysis and future reference. A reference to this historical data can be very useful in some cases. One can use the past to predict the future if, for example, network traffic is not too bursty or the network stays stable for a period of time. The data structure for historical data and each record should be dynamically created in order to save space and to archive maximum efficiency.

Another requirement for the historical data is that it should save the network data in a general way and does not favor a particular data analysis method so that

new data analysis modules can be easily added to NICAN.

3.6 The Big Picture

In summary, an outline of how NICAN works is as follows: Typically NICAN runs in the background or as a daemon process [22] waiting for a request from a network application. A distributed application calls the entry functions provided by the NICAN interface to send its request to NICAN. Upon receiving the application request, NICAN processes this request by first calling the different collector modules to gather the information needed. When the network condition change causing the monitored parameter to fall outside the bounds requested by application, the NICAN notification component sends a signal to the application. Also detailed network information is provided to the application. The adaptation process can then take place once the network information is received by the application.

Chapter 4

Implementation of NICAN

In this chapter, we discuss the implementation of NICAN. The details include all aspects of the implementation of each component of NICAN as well as the problems that arise and their solutions in the implementation. The implementation of NICAN is portable. It can be compiled and works under Unix systems with the Sun operating system or the Linux operating system.

4.1 NICAN Interface to Application

NICAN has four components: network information collection, application notification, history information table, and interface to the applications. NICAN is always running in the background or as a daemon process [22], waiting the service request from an application. An application uses the NICAN interface functions to send its request to wakeup NICAN. The request includes information on network characteristics required and the possibly bounds on a particular parameter. First the function call is made with parameters passed, then a child process is created, and the child process opens a communication channel with NICAN to interact with NICAN.

The interaction includes sending requests, receiving signals from NICAN, and processing network characteristics received with the signal. There are two reasons we choose to implement a child process to do the interaction between the application and NICAN. One is that the child process has a close relation with the application so we can use shared memory to implement data exchanging between the child and parent processes. Another reason is that the notification, specifically a signal from NICAN to the application, can now be sent to the child process. This is our method for not letting NICAN interfering with the application execution.

Figure 4.1 shows the pseudocode of the interface function `IF_FOR_APP` which takes the application request `r` as the argument for the signal handler function `SIG_HANDLER`. A shared memory `shm` can be accessed by the parent and child processes by specifying the same key value `k`. This way `shm` is attached to these processes. `C` is the communication channel between NICAN and interface. The request `r` is sent to NICAN from the child process. The remaining task of the child is to wait for the notification from NICAN. If a signal is raised by NICAN, `SIG_HANDLER` is called by the child to handle the signal. The information obtained by NICAN is read by the child and the shared memory is updated with the new data. Before the updating of `shm`, the validation of `shm` needs to be checked since shared memory is protected (see Section 4.1.2).

4.1.1 The Application Request

An application request consists of the symbolic names for the network parameters that the application wants NICAN to monitor, and possibly their ranges. Usually a distributed application is interested in the throughput and latency of the commu-

```

IF_FOR_APP (APPLICATION REQUEST r)
  if ((pid = fork()) == 0) /* child process creation */
    initialize signal handler SIG_HANDLER
    create shared memory (shm) with key (k) and attach shm to child
    create communication channel (C) with NICAN
    send r to NICAN via C
    wait for notification from NICAN
  else if (pid > 0) /* the parent process */
    create shared memory (shm) with key (k) and attach shm to parent
    return the memory to caller
  endif
END IF_FOR_APP

SIG_HANDLER
  read in characteristics value (v) from C
  if shm is ready for updating
    update the shm with v to be read by caller
    flip validation bit
  endif
END SIG_HANDLER

```

Figure 4.1: Pseudocode for NICAN interface.

```

struct app_request {
  unsigned long request;
  float        bw_up;
  float        bw_low;
  float        lat_up;
  float        lat_low;
};

```

Figure 4.2: Application Request Data Structure.

nication channel. The range of a network parameter is included if an application wants NICAN to report when the parameter falls outside of this range. For example, if an application needs to adapt the network for the effective throughput below 25,000 bps, it can include this value in the request to NICAN.

The request structure is shown in Figure 4.2. The member element `request` consists of the combination of parameters that the application wants NICAN to monitor. For example, if we define `REQ_THROUGHPUT` as `0x00000001` and `REQ_LATENCY` as `0x00000002`, `request` for both throughput and latency can be specified as

`REQ_THROUGHPUT | REQ_LATENCY`

The elements `bw_up` and `bw_low` are used to specify the bounds on throughput if applicable, i.e., `request` consists of `REQ_THROUGHPUT`. Similarly, `lat_up` and `lat_low` are defined as the bounds on latency, if applicable. More network parameters can be added by modifying this structure. Note that more definitions for the `request` element are needed in this case. No other modifications to NICAN (except for adding corresponding modules to process the new parameters) are required.

The other information in the application request includes the remote node name if point-to-point parameters are requested. If the application wants NICAN to monitor multiple remote hosts when there are two or more participating computing hosts, then their list has to be supplied to NICAN. For a distributed scientific application using Message Passing Interface (MPI), NICAN expects the application to send a rank-IP table to it. The rank is the unique identifier assigned by the MPI program for an MPI application process. The rank-IP contains the rank of each application process and the IP address of the host on which the process is running.

This table is formed by the NICAN interface and supplied to NICAN.

4.1.2 Shared Memory

Shared memory provides a fast and efficient way to process the delivered data from NICAN. When the application calls the initialization function to wake up NICAN, it creates a child process to send its request to NICAN. A chunk of shared memory is needed for the child and the parent application processes to exchange data with each other. After processing the received network information, the child process puts this information into the shared memory. The parent process, which is the application itself, examines the data in the shared memory and adapts itself based on the information. Usually, shared memory between processes is protected by using a semaphore since the information in the shared memory may only be updated by one process at a time [17]. A semaphore is a synchronization tool for protecting a critical-section. We use a simple mechanism to protect the shared memory. The first bit of shared memory is used as a validation bit. A valid bit means the information is ready for the parent process to read and the child process cannot write information at this time. An invalid bit means the information has been read by the parent process so the information is old and the child process can update the shared memory if the new information is available. The reason that this simple mechanism can be used to protect the shared memory is that only the parent process performs the read operation on the shared memory and only the child process performs the write operation on the shared memory. Although the operations on the shared memory in this case are simple, the shared memory still needs protection via this simple mechanism.

```

struct value_node {
    int type;
    float value;
};

struct shm_app {
    int signal_peak;
    struct value_node valuenode[2];
};

```

Figure 4.3: Shared Memory Structure between Application and NICAN Interface.

Figure 4.3 shows the shared memory structure. The variable `signal_peak` is the protection for this shared memory. The structure `value_node` is used for saving a single value for a parameter. The variable `type` is used to identify the type of the parameter saved.

4.1.3 Interprocess Communication

The Interprocess Communication (IPC) between the application (NICAN interface, child process) and NICAN is done by using FIFO [2]. This is a first-in first-out device for processes to pass messages to each other, and is also called a named pipe. Accessing a FIFO is the same as accessing a regular Unix file. An advantage of FIFO is that it is a full-duplex communication channel, which is desirable for NICAN since the application request and NICAN notification can share the same FIFO. A limitation of pipes [2], another mechanism for IPC, is the requirement that the communication process pairs should be related to each other such as a parent and child process. In our design, NICAN and the application do not have to have a direct relationship like this.

```

if (areq->request & REQ_LATENCY)
    coll_lat
if (areq->request & REQ_THROUGHPUT)
    coll_bw

```

Figure 4.4: NICAN Processes Application Request.

Another way to do IPC is to use a UNIX-domain socket [2, 23]. The limitation of a socket is that a system can only create a certain number of sockets during one session. If the number of sockets exceeds this limit, the communication channel cannot be created between the two processes.

4.2 Network Information Collection

The NICAN network information collection takes place dynamically while the application is executing. Currently, the collector has two modules, for collecting throughput and latency. After NICAN receives the request from the application, it first analyzes the request. The request analysis is shown in Figure 4.4. The variable `areq` is a pointer to `struct app_req` (Figure 4.2). If the request contains the latency monitoring, NICAN forks a child process to measure the latency between two end points. Using a child process to collect network characteristics has some advantages. A child process for a particular set of characteristics is created only when this information is requested by the application. The close relationship between child and parent processes makes the communication between them easy and efficient. One way to collect latency is to use UNIX utilities `ping` or `traceroute`. Both `ping` and `traceroute` use internet control message protocol (ICMP) to send an ECHO packet to remote host and calculate the round-trip-time (RTT) when the packet echoes back. The system call is used to call `ping` or `traceroute` and the results

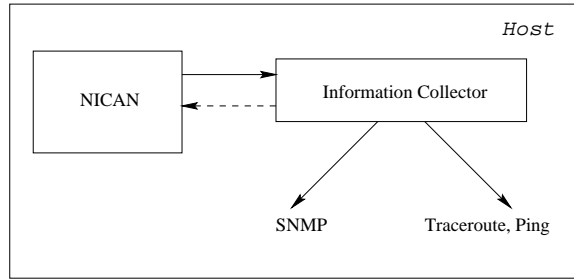


Figure 4.5: Network Information Collection.

are saved to a file in local host. Then the latency collector quickly analyzes this file and retrieves the latency. Latency is collected on a point-to-point bases. Thus, if the application wants NICAN to measure latency for more than one nodes, NICAN monitors each node in a round-robin fashion. The NICAN latency collection child process shares a chunk of memory with its parent process. They share this memory in the same way described in section 4.1.2. The present structure of the collector is shown in figure 4.5, which is a particular instance of the design features shown in Figure 3.2.

4.2.1 Information Provided by SNMP

SNMP MIB information is used for collecting the throughput. The NET-SNMP library functions are used to create the SNMP session and SNMP packet data unit (PDU), and to gather SNMP object information from an SNMP agent. The primary information we are interested in are network interface incoming octets (*ifInOctets*), outgoing octets (*ifOutOctets*), and system up time (*sysUpTime*).

The object *sysUpTime* counts the time ticks since the agent started, *ifInOctets* counts the total number of octets that the monitored host receives from a network

interface, and *ifOutOctets* counts the number of octets that this host sends out through an network interface. The network interface here is at the Data Link layer as shown on Figure 2.4, so the objects *ifInOctets* and *ifOutOctets* record all the data that goes through the interface. In a local area network (LAN) network or a cluster of computers, an individual host commonly has two interfaces: a local and an Ethernet interface. The local interface is used for processes in the same host to communicate with each other and the Ethernet interface is between the host and the outside network. In this thesis, we are only interested in the Ethernet interface information. These three objects, *ifInOctets*, *ifOutOctets*, and *sysUpTime*, are specified in one SNMP PDU. Therefore, a simple one poll (a request) to SNMP agent gets the values of the three variables at once.

If we use *time1*, *ifIn1*, and *ifOut1* to represent the values after one poll to an SNMP agent on a host, and *time2*, *ifIn2*, and *ifOut2* are the values of another poll to an SNMP agent on that host after a period of time, such that $time2 - time1$ is positive, we define the effective bandwidth as:

$$effective\ bandwidth(bps) = \frac{((ifIn2 - ifIn1) + (ifOut2 - ifOut1)) \times 800}{time2 - time1} \quad (4.1)$$

The unit for incoming and outgoing packets is octet, which is 1 byte or 8 bits. An SNMP agent reports the system up-time as hundredth second, i.e., 0.01 second.

The computer system updates the MIB table information periodically and the time period for updating depends on the host hardware and software configuration. An SNMP poll to an SNMP agent runs very fast. If the throughput collection runs

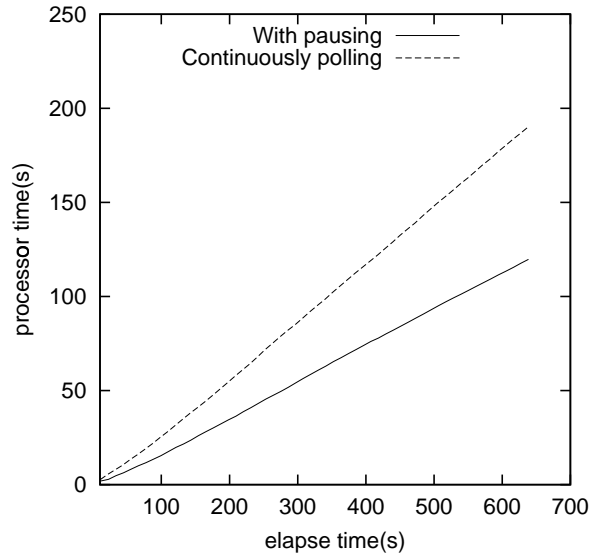


Figure 4.6: Processor Time for NICAN with and without polling pause.

continuously without pausing, most often the collection data is meaningless, either $time2 - time1$ is zero, or $(ifIn2 - ifIn1) + (ifOut2 - ifOut1)$ is zero. We found that continuously polling an SNMP agent also substantially consumes the processor execution time, which includes system and user computation time. The solution we chose is to pause the throughput collection module after a poll to an SNMP agent and before it sends the next poll. The next sleeping time is calculated from the time period of previous two successful polls divided by 2. Thus, the sleeping time depends on the system and the program adjusts the sleeping time dynamically. Figure 4.6 shows the comparison result for the processor time of NICAN with and without polling pause. Pausing the polls to SNMP agents improves the performance of NICAN.

4.3 Application Notification

After the discussion of the NICAN interface, the application notification is easy to understand. NICAN uses signal to notify an application about the events that an application is interested in.

4.3.1 Signaling Mechanism

NICAN uses a signaling mechanism to notify the applications and to provide the network information to an application. A handler is called upon signal receipt to process the information NICAN sends.

There are two approaches to implement signaling. One is to use multi-signals, in which each separate network performance parameter needs one unique corresponding signal. However, the Unix operating system (OS) provide only two types of signals, which are reserved as user signals, *SIGUSR1* and *SIGUSR2*. In general, the number of different signals in an OS is limited. Thus we have chosen an alternative approach. It consists of using only one signal to handle all information notification from NICAN. Upon receiving the signal, only one signal handler processes and analyzes the information sent by NICAN. Based on the information received, the application may perform application specific adaptations. The obvious advantage of this approach is that it does not exhaust available signals. This approach simplifies information processing and application adaptation. The function `SIG_HANDLER` in Figure 4.1 shows the pseudocode for the signal handler.

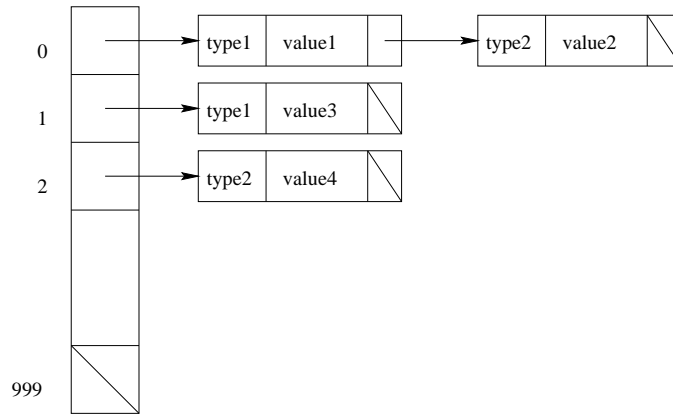


Figure 4.7: NIKAN History Table Structure.

```

ADDDATA2TABLE(TABLE t, DATA...)
  if t is not NULL pointer
    create a data list dl
    add all DATA into dl

    if t is not full
      add dl to the first empty entry of t
    else
      replace the earliest entry of t with dl
    endif
  endif
END ADDDATA2TABLE

```

Figure 4.8: Add Data to NIKAN History Table.

4.4 Historical Data Table

All information obtained by NICAN is saved into a historical table for further analysis. Each entry is a pointer to a linked list. The list contains the information about throughput, or latency, or both, depending on the availability of the information. Since it is easy to store any type of data in a list, this is the data structure chosen to implement a single entry of the table. The oldest data is overwritten by the newest information if the table is full. This is done as in a wrap-around buffer. Figure 4.7 shows the history table structure. The table in Figure 4.7 has 1000 entries. The first entry has two nodes for `type1` and `type2` parameters with values. The second entry has one node for `type1` parameter and the third one has one node for `type2` parameter. To access this table, NICAN maintains two pointers to keep track of the earliest and current data entries. Figure 4.8 shows the operation `ADDDATA2TABLE` for adding data to the table. The number of parameters passed to this function is not fixed. Thus, it is feasible to pass any number of parameters to this function. The function first creates a list `d1` and saves all the data to `d1` if the table `t` exists. The `d1` is saved into the first empty entry of `t` or the oldest entry of `t` is replaced if `t` is full.

4.5 NICAN Architecture Revisited

Figure 3.1 shows the abstract view of NICAN architecture. After the discussion of the implementation of each component of NICAN, a more detailed NICAN architecture is presented in Figure 4.9. The dashed line shows the request, signal, and information flows. Not all of the details are shown on this figure. For example, the shared memory used in the network collection is ignored. The application adap-

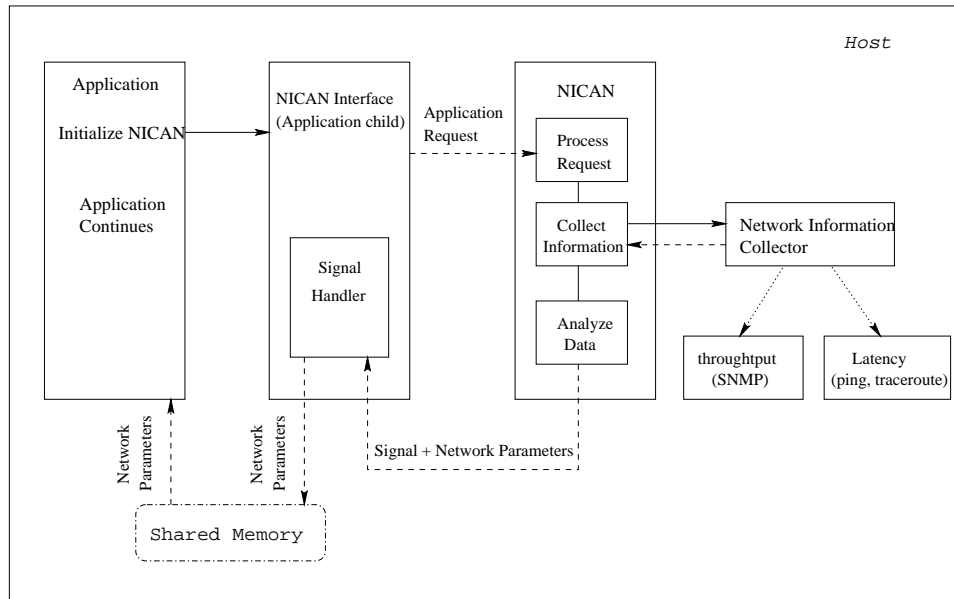


Figure 4.9: NICAN Architecture in Details.

tation is also not shown on the figure since the adaptation process is application specific.

Chapter 5

Experimental Results

In this chapter, the results of experiments testing NICAN are presented. We use NetPIPE [18] as the application to test NICAN. Both TCP- and MPI-based versions of NetPIPE are tested. An experiment on a heterogeneous computing platform is also performed and the results are analyzed.

NetPIPE [18] is a network benchmark program from Ames Laboratory (DOE), Ames, IA. The application uses a TCP connection to send a message of increasing size to measure the delivery time and then calculate the throughput. The pseudocode for this ping-pong test is shown in Figure 5.1. Alternatively, NetPIPE uses Message Passing Interface (MPI) [5] to create the connection between two hosts and measure the throughput. We must notice here that the throughput measured by NetPIPE is protocol specific. If NetPIPE uses MPI protocol, it measures the throughput that an MPI application can achieve between the two hosts. For the TCP protocol, it measures the throughput that a TCP application can achieve.

```
if it is the transmitter t then
  start timing
  send data
  end timing
else it is the receiver r
  start timing
  receive data
  end timing
endif
exchange transfer time between t and r
if it is t
  calculate bandwidth
endif
```

Figure 5.1: NetPIPE Ping-Pong Test Pseudocode.

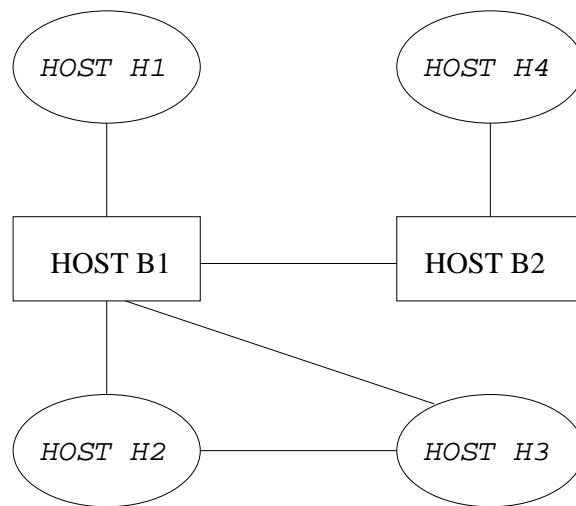


Figure 5.2: Interconnection of The Hosts Used in The Experiments.

We test NICAN in a local area network (LAN) accessed via the Ethernet protocol with the nominal bandwidth of 10 Mbps. The network is located at the Computer Science department of the University of Minnesota - Duluth (UMD). Figure 5.2 shows the interconnection of the hosts we used for testing. Hosts $H1$, $H2$, and $H3$ are SUN Solaris Sparc sun4u machines with SUN OS 5.7 Generic.106541-14 and Host $H4$ is an i686 machine with Linux OS 2.2.16-21.beosmp. $B1$ and $B2$ are two other hops that connect the hosts.

5.1 NetPIPE over TCP protocol

The following experiment and discussion are closed to [21].

In this experiment, we use the following criterion: NICAN reports the throughput to NetPIPE if (1) *the throughput is greater than 9.0 Mbps for the first time* OR (2) *the continuous changes of throughput is less than 10% for three successive measurements* [21]. The first criterion (greater than 9.0 Mbps) is considered because we are testing NICAN under a LAN environment with the Ethernet interface of 10 Mbps bandwidth. We assume that the maximum throughput is reached if the second criterion is fulfilled. The test results are shown in Figures 5.3 and 5.4.

We observe that the throughput measured by NICAN is almost always an upper bound on the throughput measured by NetPIPE (Figures 5.3 and 5.4). This can be explained by the presence of the Transport Layer Protocol (TCP) overhead in the measurements by NetPIPE. As mentioned before, NetPIPE measures the throughput for a specific protocol, while NICAN collects the information from the host Ethernet interface, which is in the Data Link layer in Figure 2.4. The difference

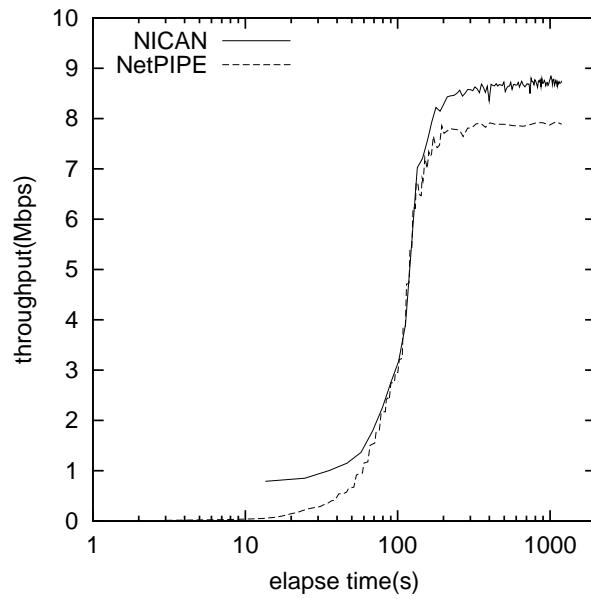


Figure 5.3: Simultaneous NetPIPE (TCP) and NIKAN throughput measurements: between Hosts H1 and H3.

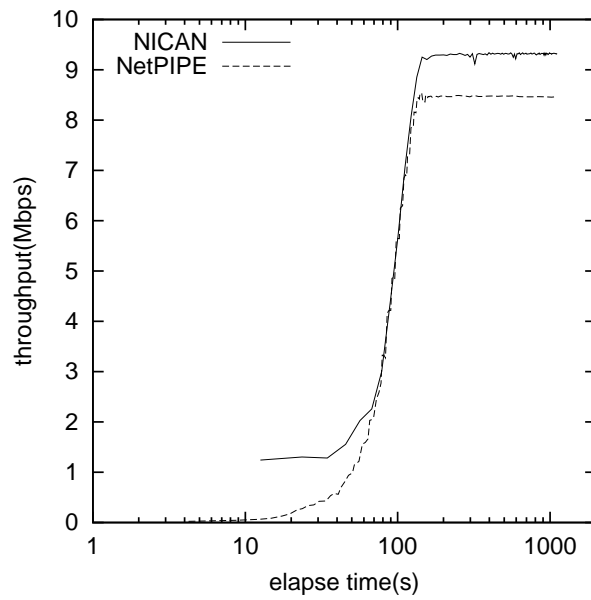


Figure 5.4: Simultaneous NetPIPE (TCP) and NIKAN throughput measurements: between Hosts H2 and H3.

between the measurements is especially pronounced at the beginning of execution, for small message sizes, and when the throughput limitations are reached for large messages which need segmentation. A message sent by the NetPIPE transmitter traverses through TCP, IP, and Data Link layers and then to the receiver. Each layer adds its header information to each message packet. For small messages and the large messages that need segmentation, the header size is significant. We also observe that the measurement difference is almost constant for these cases.

We supply NetPIPE with a notification handler to react to the peak throughput signal from NICAN. This handler also contains the adaptive features of NetPIPE. In particular, the adaptation process for NetPIPE consists of terminating the growth of the transmitted message when the notification is received. Figure 5.5 shows the performance for NetPIPE with and without invoking adaptation. Figure 5.6 shows the time to transfer a message of certain size. We notice that the throughput values measured by NICAN and NetPIPE remain very close in the two cases. At time $t_p \approx 150$ seconds, NICAN detects that the peak throughput is reached (i.e., the request criterion 1 is satisfied) and notifies NetPIPE. The time for NetPIPE to transfer a message differ greatly beyond time t_p (Figure 5.6). Thus with adaptation, NetPIPE doesn't exhaust available computer resources, while still accurately measuring the maximum throughput.

Figure 5.7 shows the latency changes during the TCP version of NetPIPE execution between H1 and H3. For small messages, latency is unchanged. At a time close to t_p , the message delay becomes larger due to the link being busy and potential congestion. We also observe the oscillation of the latency measured beyond time

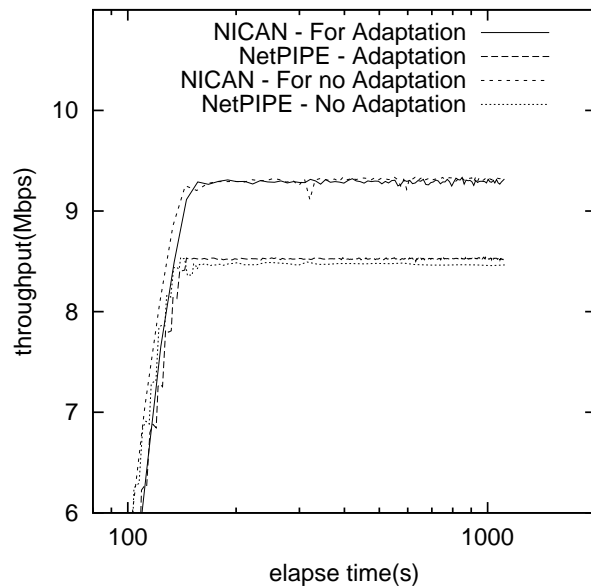


Figure 5.5: Performance of NetPIPE (TCP) with and without invoking adaptation mechanism: throughput measurements between Hosts H2 and H3.

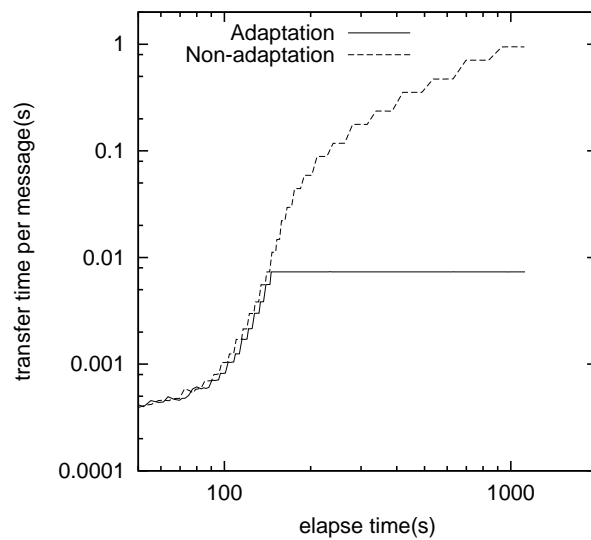


Figure 5.6: Performance of NetPIPE (TCP) with and without invoking adaptation mechanism: time to transmit a message between Hosts H2 and H3.

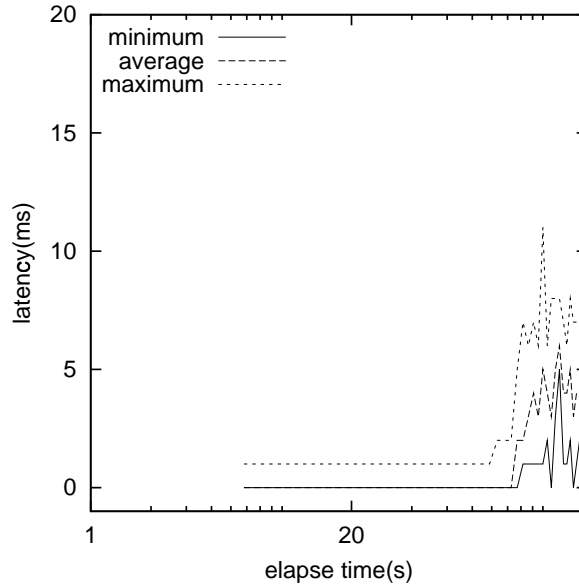


Figure 5.7: Latency measured by NICAN between hosts H2 and H3.

t_p , this is because NetPIPE does not continuously send message. When NetPIPE is calculating the throughput it measures, the link between the two hosts is empty and latency drops down to normal level.

5.2 NetPIPE over MPI

Figure 5.8 shows the throughput measured by NetPIPE and NICAN between H1 and H3. We observe that the MPI performance is far below that of TCP (see Figures 5.3 and 5.8). This is because the software overhead for MPI is greater than the one for TCP (MPI is built on top of TCP). We also notice that the throughput measured by NICAN is very close to the throughput measured by NetPIPE for the large MPI messages. This is due to the overhead of buffering large MPI messages. In particular, before using low-level network protocols, a buffer is used by MPI

program. For large messages, the buffer is not managed properly. When NICAN uses the equation 4.1 to calculate the throughput, the time difference between two polls to SNMP agent, $time2 - time1$, contains the MPI buffering delay, during which nothing is in the network communication channel. Due to the MPI buffering delay, the throughput measured by NICAN decreases. However, even in this case, the NICAN measurement is still an upper bound on the NetPIPE measurement, which is due to the protocol header size not perceived by NetPIPE.

For the experiment with NetPIPE-MPI, we modify the first criterion to notify the application when 7.0 Mbps rate is reached. The test results with and without adaptation are presented in Figure 5.9. NetPIPE-MPI with adaptation measures the maximum throughput of the link more accurately.

To support our conjecture regarding poor MPI buffering, we performed the following experiment. When the peak throughput is reached at time t_p , we start the TCP version of NetPIPE between the same two hosts, $H1$ and $H3$ with message size 8000 bytes, from which NetPIPE-TCP can fill the channel with messages. The purpose of this experiment is to fill the communication channel completely at all the time without waiting for the MPI buffer processing (During this processing time, the traffic in the channel decreases, which in turn, diminishes throughput values). The result is shown in Figure 5.10. We can notice that the throughput measured by NetPIPE decreases (compare to Figure 5.8). It takes longer than before to transfer an MPI message in the presence of a competing communicating program. The NetPIPE-MPI does not have any knowledge of another application (NetPIPE-TCP). On the other hand, NICAN measures the *total* traffic on the interface, thus possible

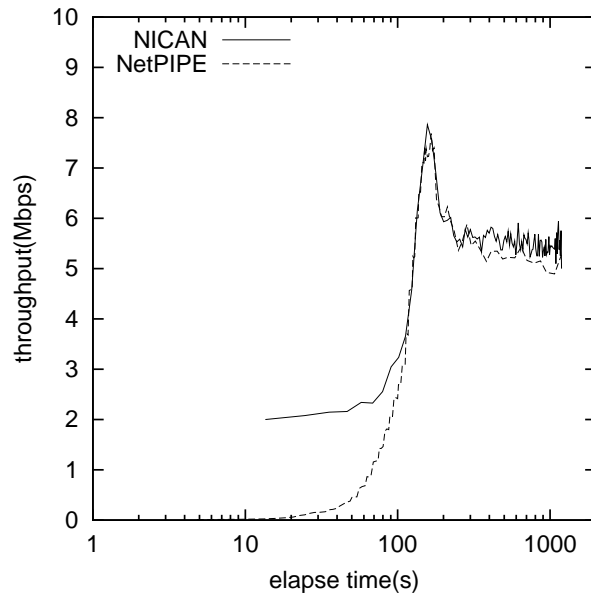


Figure 5.8: Simultaneous NetPIPE (MPI) and NICAN throughput measurements: between Host H1 and H3.

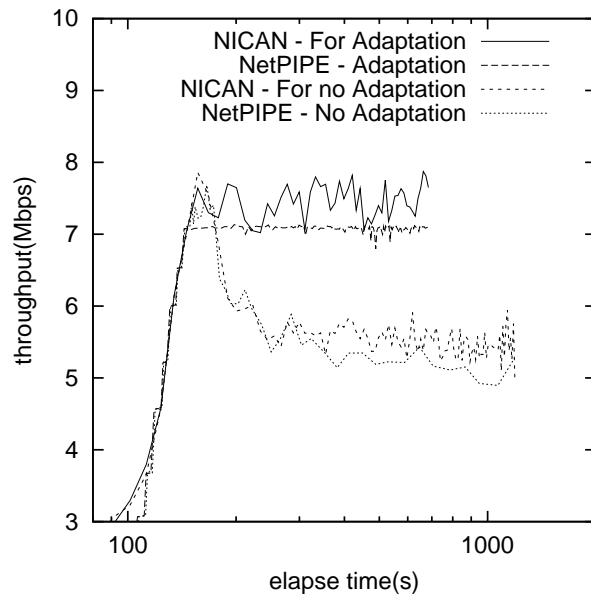


Figure 5.9: Performance of NetPIPE (MPI) with and without invoking adaptation mechanism: throughput measurements between H1 and H3.

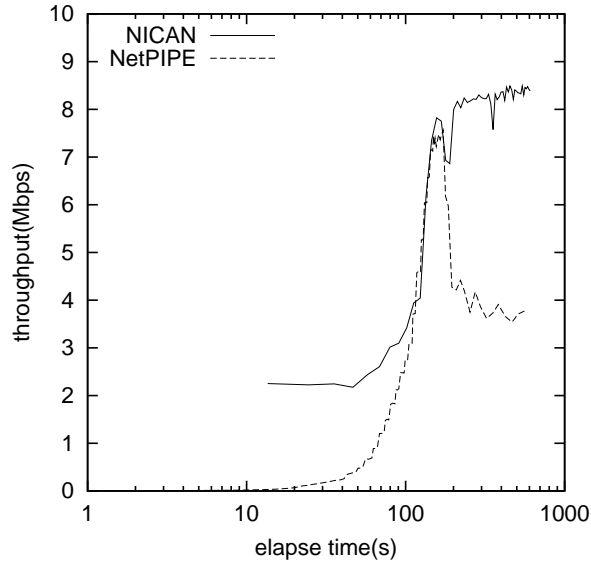


Figure 5.10: Performance of NetPIPE (MPI) and NICAN throughput measurements with competitor NetPIPE (TCP): between H1 and H3.

competitors (NetPIPE-TCP in this case) are accounted for, so that NetPIPE-MPI needs to adapt to this condition rather than to whatever it might perceive by itself.

5.3 Experiments for Heterogeneous Computing Platforms

We have conducted a preliminary set of experiments for heterogeneous computing platforms. In particular, heterogeneity is revealed in different computer architectures with different operating systems. NICAN runs on the host *H4*, which is an i686 machine with Linux OS 2.2.16-21.beosmp. Figure 5.11 shows the test results of NetPIPE-TCP communication between hosts *H1* and *H4*. The request criterion is modified to notify when 7.5 Mbps rate is reached. This criterion is chosen since

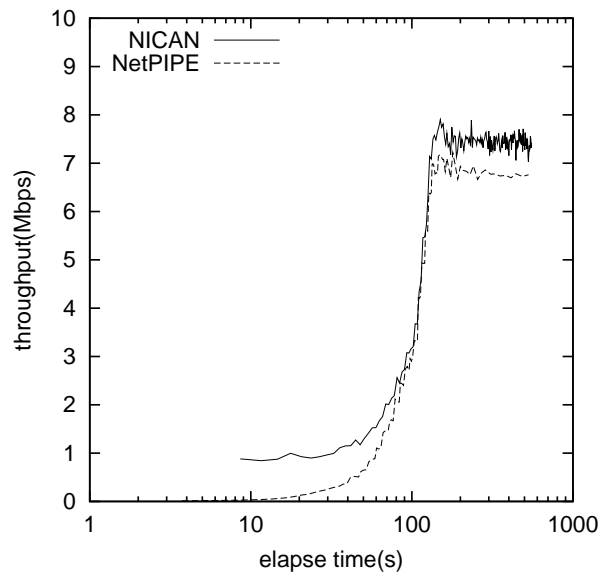


Figure 5.11: Simultaneous NetPIPE (TCP) and NICAN throughput measurements: between Hosts H1 and H4.

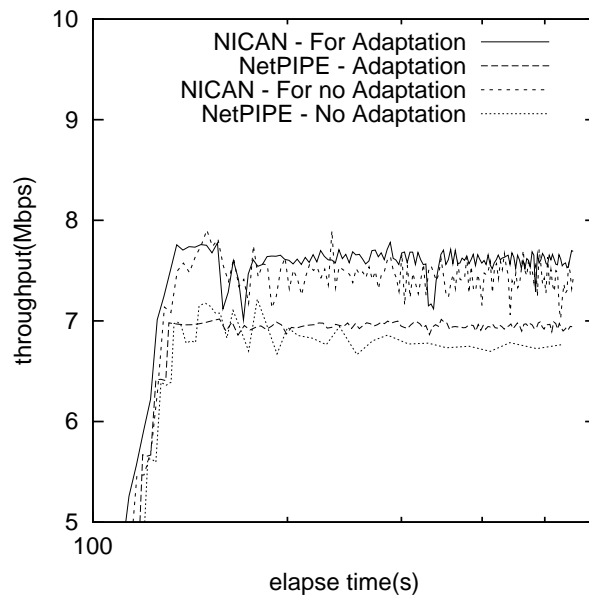


Figure 5.12: Performance of NetPIPE (TCP) with and without invoking adaptation mechanism: throughput measurements between H1 and H4.

there are two hops, $B1$ and $B2$, between $H1$ and $H4$, which decreases the throughput (compare to Figure 5.4). Figure 5.12 zooms on the interval after reaching the peak throughput. The results are similar to those between two SUN Solaris Sparc machines. This experiment shows that NICAN has the ability to handle the heterogeneous environments, which (in this case) consist of different operating systems and software configurations.

Chapter 6

Conclusions and Future Work

In this thesis we showed a simple solution for providing an application interface to distributed applications. The interface hides the underlying details of networks and provides the application with a rich set of network information at runtime. In order to demonstrate this point we designed and implemented the Network Information Collection and Application Notification (NICAN) that emphasizes simplicity of use, modularity, and a call-back application notification mechanism. Four major components of NICAN are fully implemented, these include information collection, application notification/data analysis, interface to application, and historical data table. Simplicity is the major feature of NICAN, while modular design makes NICAN extensible and versatile. Enhancements can be made by adding more modules to NICAN, such as modules for information collection and data analysis. A variety of application requests can be passed to NICAN at the initialization stage within the application. NICAN can estimate the requested network parameters and their ranges either by polling an existing network management software or benchmarking the network connection that an application uses. The call-back mechanism has been

chosen for the application notification, which uses a parameterized signal to provide feedback from NICAN to the application.

The experiments with the NetPIPE application show that the application developer can use NICAN to incorporate network characteristics into application performance tuning. Using these characteristics and an application domain knowledge, the application developer can choose an efficient adaptation process which is suitable for the specific application. The test results for the MPI version of NICAN show NICAN can interface with distributed scientific applications, which usually use MPI for interprocess communications. A heterogeneous experiment between Linux OS and Sun Solaris systems shows the portability of NICAN.

6.1 Future Work

NICAN data analysis is rather primitive now. A more extensive data analysis module can be developed such that the historical performance data is better utilized. The experiments we did are limited to a LAN network environment with Ethernet access interface. Future work can encompass heterogeneous network environments and computational grids. The testing of NICAN with user applications is in its initial stages. Our experiments with interfacing NICAN and a user application (NetPIPE) produced promising results. However, the applicability of NICAN has to be tested with a real-world application, such as PPARSLIB[16]. Studying application adaptation capabilities and ways to set the ranges for the requested parameters is another possible extension of this research.

Bibliography

- [1] IEEE 802.3, Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, 1998.
- [2] D. Curry. *UNIX Systems Programming for SVR4*. O'Reilly & Associates, Inc., 1996.
- [3] T. Dewitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. ReMoS: A resource monitoring system for network aware applications. Technical Report CMU-CS-97-194, School of Computer Science, Carnegie Mellon University, 12 1997.
- [4] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 7 1998.
- [5] R. Hempel. The MPI standard for message passing. In W. Gentsch and U. Harms, editors, *High-Performance Computing and Networking, International Conference and Exhibition, Proceedings, Volume II: Networking and Tools*, volume 797, pages 247–252, 1994.
- [6] M. Hemy, U. Hengartner, P. Steenkiste, and T. Gross. MPEG system streams in best-effort networks. In *Packet Video'99*, 4 1999.

- [7] R. Jain. *FDDI Handbook: High-Speed Networking Using Fiber and Other Media*. Addison Wesley Longman, Inc., 1994.
- [8] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for networkaware applications. In *Cluster Computing, no. 2*, pages 139–151, 1999.
- [9] N. Miller and P. Steenkiste. Collecting network status information for network-aware applications. In *INFOCOM (2)*, pages 641–650, 2000.
- [10] NET SNMP project. Web Site, <http://net-snmp.sourceforge.net/>.
- [11] B. Noble. *Mobile Data Access*. PhD thesis, Carnegie Mellon University, 1998.
- [12] W. Norton and A. Adams. Project netscarf. *ConneXions*, 10(7), 7 1996.
- [13] L. Peterson and B. Davie. *Computer Networks: A System Approach*. Morgan Kaufmann Publishers, Inc., second edition, 2000.
- [14] M. Rose. Management information base for network management of TCP/IP-based internets: MIB-II, 3 1991. RFC-1213.
- [15] M. Rose and K. McCloghrie. Structure and identification of management information for TCP/IP-based internets, 5 1990. RFC-1155.
- [16] Y. Saad, G. Lo, and S. Kuznetsov. *PSPARSLIB Users Manual: A Portable Library of Parallel Sparse Iterative Solvers*. Department of Computer Science, University of Minnesota, 1998.
- [17] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison Wesley Longman, Inc., fifth edition, 1998.

- [18] Q. Snell, A. Mikler, and J. Gustafson. NetPIPE: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, 6 1996.
- [19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. D'Ongarra. *MPI - The complete Reference*, volume 1. The MIT Press, second edition, 1998.
- [20] M. Sosonkina. Runtime adaptation of an iterative linear system solution to distributed environments. In *Applied Parallel Computing, PARA'2000*, volume 1947 of *Lecture Notes in Computer Science*, pages 132–140, Berlin, 2001. Springer-Verlag.
- [21] M. Sosonkina and G. Chen. A simple tool for providing dynamic network information to an application. In *Parallel Computing Technologies Sixth International Conference (PaCT-2001)*, 2001.
- [22] R. Stevens. *UNIX Network Programming Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall, Inc., second edition, 1997.
- [23] R. Stevens. *UNIX Network Programming Interprocess Communications*, volume 2. Prentice-Hall, Inc., second edition, 1998.
- [24] M. Subramanian. *Network Management: Principles and Practice*. Addison Wesley Longman, Inc., 2000.
- [25] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *IEEE High Performance Distributed Computing (HPDC-7)'98*, pages 28–31, 7 1998.

- [26] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *6th IEEE Symp. on High Performance Distributed Computing*, pages 316–325, 1997.

